Question 6: Cache in While You Can (17 points, 26 Minutes)

Consider a single 4KiB cache with 512B blocks and a write-back policy. Assume a 32-bit address space.

a) If the cache were direct-mapped, (1 pt each)
   \[ \frac{4}{2^9} = 8, \log_2(512) = 9. \]
   
   \# of rows? 8
   \# of offset bits? 9

b) If the cache were 4-way set associative,
   \[ 8/4 = 2, \log_2(2) = 1. \]
   \[ 32 – 1 – 9 = 22. \]
   \[ 512^8 + 22 + 2 = 4120. \]

   # of tag bits? 22
   # of index bits? 1
   # of bits per cache slot? 4120

   \[ 512^8 \text{ is data (blocks), 22 is tag, 2 is Valid + Dirty bits.} \]

Consider an array of the following location structs:

```c
typedef struct {
    ... // some undefined number of other struct members
    int visited;
    int danger;
} location;

location locs[NUM_LOCS];
```

Here's a piece of code that counts the number of places we've visited. Assume this gets executed somewhere in the middle of our program, that count is held in a register, and the size of the array is greater than 4 KiB.

```c
for(int i = 0; i < NUM_LOCS; i++)
    if(locs[i].visited) count++;
```

c) What's the fewest possible number of bytes written to main memory? (1 pt) 0 B

d) What's the greatest possible number of bytes written to main memory? (1 pt) 4 KiB

We're reading, not writing. What will be written back are dirty blocks already in the cache.

Now consider if we store the visited and danger information in individual arrays instead:

```c
int visited[NUM_LOCS];
int danger[NUM_LOCS];
```

-0.5 pt if missing the word “locality.”

e) This way, the cache can exploit better spatial locality for the above task. (1 pt)

   We can expect a lower (higher or lower) miss rate (1 pt)

   because of the change in the number of compulsory (type of cache miss) misses. (1 pt)
   (also accepted: read)
Consider the following code with $\text{NUM_LOCS} > 2^{10}$.

```c
for(int i = 0; i < \text{NUM_LOCS}; i++)
    \quad \text{if}(\text{visited}[i] \&\& \text{danger}[i] > 5) \text{count}++;
```

Two memory accesses are made per iteration: one into `visited`, the other into `danger`. Assume that the cache has no valid blocks initially. **You are told that in the worst case, the cache has a miss rate of 100%**. Consider each of the following possible changes to the cache individually.

f) **Mark each as E, if it eliminates the chances of this worst-case scenario miss rate, R if it reduces the chances, or N if it’s not helpful. (2 pts each)**

- More sets, same block size, same associativity  \hspace{1cm} \_\_R\_\
- Double associativity, half block size, same total cache size  \hspace{1cm} \_\_E\_\
- Everything stays the same but use a write-through policy instead  \hspace{1cm} \_\_N\_

Given that the worst case miss rate is 100% for blocks that hold more than 1 piece of array data, our cache **must** be direct-mapped. In addition, the worst case happens when the `visited` and `danger` arrays start in blocks that map to the same row AND have the same offset.

- With more sets/rows, we are increasing the size of the cache. If the cache size changes such that addresses of `visited[i]` and `danger[i]` no longer map to the same row, then we no longer have the worst case scenario. This is not guaranteed to happen, so the chances are reduced.
- Increasing associativity completely removes the ping-pong effect.
- A write-through policy does not change the behavior of the cache at all.
**Question 3: Caches**

a) Block size $B$ is 16 bytes, so $\log_2(16) = 4 = O$. The 32 KiB cache (C) holds $2^{15}/2^4 = 2^{11}$ blocks, so $\log_2(2^{11}) = 11 = I$. Then we are given 32 address bits, so $T = A - I - O = 32 - 11 - 4 = 17$.  

$$17:11:4$$ (1 pt)

b) Bits per row for a direct-mapped cache is \( V + D + T + 8*8 = 1 + 0 + 17 + 8*16 = 146 \) bits/row (1 pt)  

In this case, $D = 0$ since we are using a write-through policy.

c) Every access inside Loop 1 is a miss, since $\text{OFFSET}*\text{sizeof(int)}$ is equal to the size of the cache.

Therefore, Hit rate for loop 1: $0\%$ (1 pt)  

Types of misses:  
- Compulsory (0.5 pt)  
- Conflict (0.5 pt)

(d) Notice that in the innermost loop, we call `rand` 4 times, but with the same arguments (a range of width 4). Since our array is aligned with a block boundary, every single iteration of the outer loop follows this pattern (where $a$, $b$, $c$, $d$ are the 4 iterations of the inner loop that happen once every iteration of the outer loop):  
   a. Miss (loads the block into memory)  
   b. Hit  
   c. Hit  
   d. Hit  

Thus, we have a hit rate of 75% (1 pt)  

Types of Misses: conflict (1 pt)  

(-0.5 pts if extra/incorrect Cs were given)

e) Modifying our cache to be 2-way set associative creates two “slots” in the cache for each index. This solves all of the conflicts we had in Loop 1, and thus both sets of 32 ints are located in the cache after the completion of Loop 1. This leads to a hit rate of 100% for loop 2. (3 pts)

f) Removing the line labeled ACCESS #2 is another way of removing all conflicts from our code. All values from A that are accessed in Loop 1 remain in the cache after Loop 1 completes, leading to a hit rate of 100% for loop 2. (3 pts)

g) We assume that the functionality of this program is to store 64 ints and then randomly print 4 per block from the first set of 32 ints. Thus, we may ignore all of the “junk” that is located between indices [32, 8191] in A. Since we don’t care about these values, we can reduce OFFSET to 32. This entirely eliminates the “junk” in between the values we care about. All of the counts are now stored in [0, 31] and all of the (count + count)s are stored in [32, 63]. Now the size of our array is far less than that of the cache, and thus the entire array will fit into the cache at once. Therefore, shrinking OFFSET to 32 will increase our Loop 2 hit rate to 100%, which is the maximum possible value. Since the wording could be interpreted in two different ways, points were also given for “shrinking OFFSET by 32,” which achieves the same goal. (3 pts)
M2) Cache Money, y’all (10 pts)

The key to this problem was analyzing the memory access pattern of SwapLeft. For every index $i$, SwapLeft performed (in order) a read from $A$, read from $B$, write to $A$, then write to $B$. So that’s 4 memory accesses per byte (since data is type `uint8_t`) in strictly alternating fashion.

a) Best-case scenario for a direct-mapped cache is no conflict misses (i.e. $A[i]$ and $B[i]$ map to different rows). Since we access bytes in memory sequentially (i++), for every cache block for $A$ or $B$, we get an initial miss then hit on the block boundary, followed by 2 hits for the rest of the bytes in the block ($a-1$ bytes if block size is $a$). So in total this leaves us with a best ratio of $2*(a-1)+1:1 = H:1$. Solving, we get $2a-1=H$, so $a=(H+1)/2$. (2 pts)

1 pt if answered $H+1$.

b) Worst-case scenario is that $A$ and $B$ live at addresses that conflict (i.e. $A[i]$ and $B[i]$ always map to the same cache row). Because we alternate accesses between the arrays, we always conflict in the cache and we never get a hit, so the worst ratio is $0:<\text{any non-zero number}>$. (1 pt)

c) For swapping, we must read and write from both $A$ and $B$. To improve the worst case cache performance, we need to guarantee that we access one of the arrays consecutively. The two solutions are: read $A$, read $B$, write $B$, then write $A$ and read $B$, read $A$, write $A$, then write $B$. Both of these involved using both temporary variables given to you ($\text{tmpA, tmpB}$). (1 pt)

d) Same worst case scenario as part (b) with conflicting addresses of $A[i]$ and $B[i]$. But our new access pattern for SwapRight generates MMHM on the cache block boundary, followed by HMHM for the remaining $a-1$ bytes of the block. Notice the compulsory miss on the first byte that actually becomes a hit in the remaining bytes because of the last write from the previous byte. This means our ratio for a full cache block is $1+2*(a-1):3+2(a-1)$, which simplified to $2a-1:2a+1$. (2 pts)

e) Moving to 2-way set associative, SwapLeft only accesses two arrays, so even if $A[i]$ and $B[i]$ map to the same set, they can both co-exist in the cache. With LRU, the read from $B$ cannot kick out the block of $A$, regardless of whether the cache is empty or full, so we end up with the same cache performance as part (a), where we had $2*(a-1)+1:1 = 2a-1:1$. (2 pts)

1 pt if answered $a-1:1$

1 pt if answered $2a:1$

MRU works essentially the same as direct-mapped once the cache is full (the LRU block in each set will remain there until the cache gets flushed). This leads us back to our worst-case scenario from part (b), which is $0:<\text{any non-zero number}>$. (2 pts)
d) **3 points.** Ash Ketchum has six slots in his party, each of which can hold a single Pokémon. Additionally, Ash has access to a PC (personal computer) which holds the rest of the Pokémon he owns. Essentially, his party acts as a “cache” for accesses to the PC (the “memory”).

i. Each slot in Ash’s party can hold any Pokémon. What kind of cache is this analogous to? (Circle one)

- Set-associative
- Write-back
- **Fully Associative**
- Direct Mapped
- Write-through

**+1 point for circling the correct type.**

ii. Ash’s party exploits **temporal** locality but not **spatial** locality.

+0.5 points for correctly-filled “temporal locality” blank.

+0.5 points for correctly-filled “spatial locality” blank.

*Explain in one sentence (the answer below is more than one sentence for clarity):*

The party has “one-unit” slots and thus does not exhibit spatial locality; we don’t pull any extra pokemon into the party when we make a request for a pokemon (effective block size of one). On the other hand, a fully associative cache will likely use some kind of LRU scheme, which takes advantage of temporal locality.

**+0.5 points for identifying that fully-associative cache often holds recently-used elements.**

**+0.5 points for stating that the block size of one doesn’t exploit spatial locality.**

---

e) **3 points.** Fill in the *recursive* definition of `strcmp()` below. `strcmp` returns 0 if the contents of the strings are identical, otherwise it returns the difference in the ASCII representations of the first non-matching characters (negative number if $s_1 < s_2$ lexicographically). *(Clarification from when the exam was given: `strcmp` will never be called with $s_1=\text{NULL}$ or $s_2=\text{NULL}$)*

```c
int strcmp(char *s1, char *s2) {
    if ( *s1 == *s2 && *s1 != '\0' )
        return strcmp( s1+1, s2+1 );
    return (*s1 - *s2);
}
```

+1 point for correctly comparing *s1 and *s2 in condition (or any working variation).
+1 point for correctly checking for the end of the input string(s) (only one check is necessary).
+1 point for a correct recursive call in the return statement.

−0.5 points for using the pointers s1 and s2 where elements should have been used or vice-versa

*NOTE: significant variations from the structure of this answer that were fully correct or close to correct in concept were individually graded. If fully correct, they received the full three points; otherwise, they were subject to the holistic grading standard set by this rubric.*

---

f) **8 points total.** MIPS already has a 64-bit architecture, so it’s just a matter of time before MIPS128 is released. Let’s help them out with their 128-bit instruction format design. Answer the following questions *independently* based on the MIPS32 design taught in class.

−1 point from the whole question if all three parts were answered for a 64-bit instruction format.
Question 4: Caches (11 pts)

We have a 64 KiB address space and two possible data caches. Both are 1 KiB, direct-mapped caches with random replacement and write-back policies. Cache X uses 64 B blocks and Cache Y uses 256 B blocks.

a) Calculate the TIO address breakdown for Cache X: [1.5 pts]

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

b) During some part of a running program, Cache Y's management bits are as shown below. Four options for the next two memory accesses are given (R = read, W = write). Circle the option that results in data from the cache being written to memory. [2 pts]

<table>
<thead>
<tr>
<th>Slot</th>
<th>Valid</th>
<th>Dirty</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1000 01</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>0101 01</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1110 00</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0000 11</td>
</tr>
</tbody>
</table>

(1) R 0x4C00, W 0x5C00  (2) W 0x5500, W 0x7A00
(R then W into slot 00) (W into dirty slot 01 – tag matches, W into slot 10)

(3) W 0x2300, R 0x0F00  (4) R 0x3000, R 0x3000
(W into slot 11, then kick dirty block out) (2 reads into non-dirty slot 00)

(1) R then W into slot 00 line
(2) W into dirty slot 01 – tag matches, W into slot 10 line
(3) W into slot 11, then kick dirty block out line
(4) R then W into slot 00 line

Access pattern is R then W for each address. To get a hit rate of 15/16, need to access exactly 8 addresses per block (compulsory miss on first R, then followed by all hits). Since block size for Cache Y is 256 B and char size is 1 B (256 array elements per block), we need our LEAP to be 256/8 = 32.

c) The code snippet below loops through a character array. Give the value of LEAP that results in a Hit Rate of 15/16 for Cache Y. [4 pts]

```c
#define ARRAY_SIZE 8192
char string[ARRAY_SIZE];   // &string = 0x8000
for(i = 0; i < ARRAY_SIZE; i += LEAP)
    string[i] |= 0x20;   // to lower
```

Access pattern is R then W for each address. To get a hit rate of 15/16, need to access exactly 8 addresses per block (compulsory miss on first R, then followed by all hits). Since block size for Cache Y is 256 B and char size is 1 B (256 array elements per block), we need our LEAP to be 256/8 = 32.

d) For the loop shown in part (c), let LEAP = 64. Circle ONE of the following changes that increases the hit rate of Cache X: [2 pts]

- Increase Block Size (hit rate ↑)
- Increase Cache Size (no change to hit rate)
- Add a L2$ (miss penalty ↓)
- Increase LEAP (hit rate ↓)

e) For the following cache access parameters, calculate the AMAT. All miss and hit rates are local to that cache level. Please simplify and include units. [1.5 pts]

<table>
<thead>
<tr>
<th>L1$ Hit Time</th>
<th>L1$ Miss Rate</th>
<th>L2$ Hit Time</th>
<th>L2$ Hit Rate</th>
<th>MEM Hit Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 ns</td>
<td>40%</td>
<td>20 ns</td>
<td>95%</td>
<td>400 ns</td>
</tr>
</tbody>
</table>

AMAT = 2 + 0.4 * (20 + 0.05 * 400)  AMAT = 2 + 0.4 * 400

18 ns 162 ns
We are using a 20-bit byte addressed machine. We have two options for caches: Cache A is fully associative and Cache B is 4-way set associative. Both caches have a capacity of 16 KiB and 16 B blocks.

a) Calculate the TIO address breakdown for Cache A: [1 pt]

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

b) Below is the initial state of one set (four slots) in Cache B. Each slot holds 2 LRU bits, with 0b00 being the most recently used and 0b11 being the least recently used. Circle ONE option below for two memory accesses that result in the final LRU bits shown and only one block replacement. [2 pt]

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slot</td>
<td>LRU bits</td>
</tr>
<tr>
<td>Index</td>
<td>0000 0010 10</td>
</tr>
<tr>
<td>0</td>
<td>0110 1010 00</td>
</tr>
<tr>
<td>1</td>
<td>0000 0001 10</td>
</tr>
<tr>
<td>2</td>
<td>0101 0101 01</td>
</tr>
<tr>
<td>3</td>
<td>1010 1100 11</td>
</tr>
</tbody>
</table>

Options:
(1) 0x019D0, 0xAD9D0
(2) 0xAC9E0, 0x129E0
(3) 0xAD9D0, 0x019D0
(4) 0x129E0, 0xAC9E0

(1) 0x019D0, 0xAD9D0
(2) 0xAC9E0, 0x129E0
(3) 0xAD9D0, 0x019D0
(4) 0x129E0, 0xAC9E0

For the code given below, calculate the hit rate for Cache B assuming that it starts cold. [3 pt]

```c
#define ARRAY_SIZE 8192
int int_arr[ARRAY_SIZE];    // &int_arr = 0x80000
for (int i = 0; i < ARRAY_SIZE / 2; i++) {
    int_arr[i] *= int_arr[i + ARRAY_SIZE / 2];
}
```

Access pattern is R, R + ARRAY_SIZE/2, W, i. Array index jump is 4096*4 = 2^{14} B away, so maps into same set same set (I+O=12<14).
N=4, so both blocks can fit in cache at once. Indices are not revisited and each block holds 16 B / 4 B = 4 indices, so first index is MMH, other 3 are HHH, so HR = 10/12 = 5/6.

d) For each of the proposed changes below, write U for “increase”, N for “no change”, or D for “decrease” to indicate the effect on the hit rate of Cache B for the loop shown in part (c): [2 pt]

<table>
<thead>
<tr>
<th>Proposed Change</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct-mapped</td>
<td>D</td>
</tr>
<tr>
<td>Double ARRAY_SIZE</td>
<td>N</td>
</tr>
<tr>
<td>Random block replacement</td>
<td>D</td>
</tr>
</tbody>
</table>

e) Calculate the AMAT for a multi-level cache given the following values. Don’t forget units! [2 pt]

```
\[
\text{HT}_1 + \text{MR}_1 \times \text{HT}_2 + \text{GMR} \times \text{HT}_{\text{MEM}} = 4 \times 0.2 + 5 \times 0.05 + 25 \times 0.05 = 104 \text{ ns}
\]
```